DTIC FILE COPY

DTIC QK

85-0080

AD-A221 796

Rochester

Department of Computer Science
University of Rochester
Rochester, New York 14627

DTIC
ELECTE
MAY 15 1990
S E D

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

90 05 14 113

The HORNE Reasoning System

James F. Allen, Mark Giuliano, and Alan M. Frisch
Computer Science Department
University of Rochester
Rochester, NY 14627

TR 126 revised
September 1984

Abstract

HORNE is a programming system that offers a set of tools for building automated reasoning systems. It offers three major modes of inference:

-- a horn clause theorem prover (backwards chaining mechanism);

-- a forward chaining mechanism; and

-- a mechanism for restricting the range of variables with arbitrary predicates.

All three modes use a common representation of facts, namely horn clauses with universally quantified variables, and use the unification algorithm. Also, they all share the following additional specialized reasoning capabilities: 1) variables may be typed with a fairly general type theory that allows intersecting types; 2) full reasoning about equality between ground terms, and limited equality reasoning for quantified terms; and 3) escapes into LISP for use as necessary. This paper contains an introduction to each of these facilities, and the HORNE User's Manual.

This work was supported in part by the Defense Advanced Research Projects Agency under Grant N00014-82-K-0193, the Office of Naval Research under Grant N00014-80-C-0197, and the National Science Foundation under Grant IST-8210564.

DO NOT REMOVE
>*%UAAAAAA34164780*

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR 126 revised	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) The HORNE Reasoning System		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) James F. Allen, Mark Giuliano, and Alan M. Frisch		8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0197 N00014-82-K-0193
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department University of Rochester Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS DARPA 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE September 1984 13. NUMBER OF PAGES 57
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

logic programming, equality reasoning, type reasoning, inference, backwards chaining, forward chaining, constraints

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

HORNE is a programming system that offers a set of tools for building automated reasoning systems. It offers three major modes of inference:

- a horn clause theorem prover (backwards chaining mechanism);
- a forward chaining mechanism; and
- a mechanism for restricting the range of variables with arbitrary predicates.

All three modes use a common representation of facts, namely horn clauses

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Abstract (cont.)

with universally quanitified variables, and use the unification algorithm. Also, they all share the following additional specialized reasoning capabilities: 1) variables may be typed with a fairly general type theory that allows intersecting types; 2) full reasoning about equality between ground terms, and limited equality reasoning for quantified terms; and 3) escapes into LISP for use as necessary. This paper contains an introduction to each of these facilities, and the HORNE User's Manual.

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist
A-1

1. Introduction

This is a brief introduction to the major reasoning modes and facilities provided by the HORNE reasoning system. Details on the actual system are contained in the HORNE User's Manual which forms the second half of this report. In this section, we will first discuss the basic reasoning modes, and then outline the specialized reasoning systems embedded in HORNE.

2. The Basic Reasoning Modes

There are three basic reasoning modes. The first two correspond to the antecedent and consequent theorem mechanisms of PLANNER, and are called forward chaining and backward chaining, respectively. The third is most closely related to reasoning with constraints, and is called constraint posting.

Independent of the mode of reasoning, all facts are in the form of horn clauses, which can be viewed as logical implications with a single consequent. Thus

P < Q

read as "if Q then P," is a horn clause, as is

P <

which simply asserts P, and as is

P < Q, R

which should be read as "if Q and R, then P." The following is *not* a horn clause, because there are two consequences:

* P, Q < R.

Note that, in more general systems of this type, this would be read as "if R, then P or Q."

A horn clause may contain globally scoped, universally quantified variables which are indicated by a prefix of "?". Thus

(P ?x) < (Q ?x)

is a horn clause that is read as "for any x, if Q of x holds, then P of x holds." Finally, whenever the process of matching two formulas is discussed, we are referring to the full unification algorithm found in resolution theorem-proving systems extended to unify lists in LISP format. This extension is explained in detail in the HORNE User's Manual.

2.1 Backwards Chaining

This mode provides a PROLOG-like theorem prover. It searches a horn clause that could prove the given goal, and attempts to prove the antecedents of the horn

clause. It uses a depth-first, backtracking search. For the reader not familiar with such systems, see [Kowalski, 1979].

As an example, consider the following axioms:

All fish live in the sea
(1) (LIVE-IN-SEA ?x) < (FISH ?x)
All Cod are fish.
(2) (FISH ?x) < (COD ?x)
All Mackerel are fish.
(3) (FISH ?x) < (MACKEREL ?x)
Whales live in the sea.
(4) (LIVE-IN-SEA ?y) < (WHALE ?y)
Homer is a Cod.
(5) (COD HOMER) <
Willie is a Whale.
(6) (WHALE WILLIE) <

Given these axioms, we can prove Willie lives in the sea as follows, using a straightforward *backtracking search*. We have the goal:

(7) (LIVE-IN-SEA WILLIE)

Rule 1 appears applicable: Unifying (1) with (7) we get

(LIVE-IN-SEA WILLIE) < (FISH WILLIE)

So we have a new subgoal:

(8) (FISH WILLIE)
Rule (2) applies, giving
(FISH WILLIE) < (COD WILLIE),
so we have a new subgoal
(9) (COD WILLIE)
× No rule applies, try (8) again.
Rule (3) applies, giving
(FISH WILLIE) < (MACKEREL WILLIE)
So we have a new subgoal
(10) (MACKEREL WILLIE)
× No rule applies, try (8) again, no more ways to prove (8)
× No rule applies, try (7) again
Rule (4) applies giving
(LIVE-IN-SEA WILLIE) < (WHALE WILLIE)
So we have a new subgoal
(11) (WHALE WILLIE)
Rule (6) asserts (11) as a fact
✓ Goal (11) is Proved.
✓ Goal (7) is Proved.

2.2 Forward Chaining

The rules for forward chaining are quantified horn clauses augmented with a *trigger*. Such a rule is applied whenever a fact is added that matches (i.e., unifies with) the trigger. In such a case, the reasoner attempts to prove the antecedents of the rule and, if it is successful, asserts the consequence. In general, each of the antecedents is attempted by simple data base lookup only. In other words, the backwards chaining reasoner is not invoked to prove an antecedent. There is an option, however, to invoke the backwards reasoning if desired.

For example, consider maintaining the simple transitive relation < (less than) using forward chaining. The axiom we want to use to ensure the complete DB is

$$\forall\ x,y,z\ \ LT(x,y)\ \&\ LT(y,z) \supset LT(x,z).$$

To implement this using forward chaining rules, we have the following:

	Trigger	**Rule**
(12)	(LT ?x ?y)	(LT ?x ?z) < (LT ?x ?y) (LT ?y ?z)
(13)	(LT ?y ?z)	(LT ?x ?z) < (LT ?y ?z) (LT ?x ?y)

Consider the following additions:

(LT B C) triggers rules (12) and (13), but nothing can be proved

(LT A B) triggers (12) ?x ← A, ?y ← B
proves (LT A B) ✓
proves (LT B ?z), ?z ← C
adds (LT A C)
triggers (12) ?x ← A, ?y ← C
proves (LT A C)
fails on (LT C ?z)
triggers (13) ?y ← A, ?z ← C
proves (LT A C)
fails on (LT ?x A)
triggers (13) ?y ← A, ?z ← B
proves (LT A B)
fails on (LT ?x A)

As one can see, the rules apply recursively on inferred additions, and the search space generated by the forward chaining rules is completely searched. The forward chainer detects possible infinite loops that could result from adding the same fact twice.

2.3 Constraint Posting

The last facility allows proofs of goals to be delayed for certain predicates until more is known about the arguments to the predicate. In particular, it allows one to

delay proving a formula until one of its variables is bound.

This is best illustrated by example. Assume we want to define a predicate of two arguments, ?x and ?y, that is true iff ?x and ?y are bound to different terms. The most common way to implement this in PROLOG systems is to use negation by failure on the EQ predicate, which is simply defined by

(14) (EQ ?x ?x)

Thus EQ forces two terms to unify, and fails if they cannot. Using this, they define

(15) (NEQ* ?x ?y) < (UNLESS (EQ ?x ?y))

where UNLESS is negation by failure. This formulation gives undesirable results when one of its terms is unbound. In particular, it binds a variable argument to make the terms equal. Thus with the axioms

(16) (P ?x ?y) < (NEQ* ?x ?y) (R ?y)

(17) (R B)

we could not prove (P A ?y) for the predicate (NEQ* A ?y) would fail since (EQ A ?y) succeeds by binding ?y to A.

To avoid this, we could define NEQ* so that it only fails when both arguments are bound. But this would allow incorrect proofs as the variable could later be bound violating the distinctness condition. What is needed is a facility to delay the evaluation of (NEQ* ?x ?y) until both arguments are bound. We do this by a mechanism called *posting*.

If a literal is POSTED and contains no variables, it is treated as a usual literal. The proof succeeds or fails and the posting has no effect. If the literal does contain a variable, the evaluation of that literal is delayed until the variable is bound. Thus we define a new predicate DISTINCT by

(18) (DISTINCT ?x ?y) < (POST (NEQ* ?x ?y)).

Now, using a modified axiom (16), namely,

(19) (P ?x ?y) < (DISTINCT ?x ?y) (R ?y)

and the modified definition of NEQ* as in axioms (20) - (22),

(20) (NEQ* ?x ?y) < (unbound ?x)

(21) (NEQ* ?x ?y) < (unbound ?y)

(22) (NEQ* ?x ?y) < (NOT (EQ ?x ?y))

we can prove (P A ?y), resulting in ?y being bound to B as follows:

Goal: (P A ?y)

Subgoals: (DISTINCT A ?y) (R ?y)

(DISTINCT A ?y) is proven using (18), but the subgoal (NEQ* A ?y) is not evaluated in the normal manner since ?y is unbound. Instead, the call succeeds and ?y is annotated to be NEQ* from A.

(R ?y) succeeds from axiom (17) if ?y can be bound to B. The unifier checks (NEQ* A B), which succeeds, allowing ?y to be bound.

Goal proved is (P A B). Note that DISTINCT is a built-in predicate in HORNE and is defined using this mechanism.

Let us consider this mechanism in a bit more detail. After a literal has been POSTED, its variables are annotated using a form such as

(any ?x (Q ?x))

which is a term that will unify with any term such that Q holds for that term. Thus (any ?x (Q ?x)) unifies with A only if we can prove (Q A).

If there are multiple variables in a posting, each variable is annotated separately, and the constraints on each are checked as each is bound. For example, the trace of the proof of (P ?x ?y) given axioms (17) - (22) is as follows:

Goal: (P ?x ?y)

Rule (19) applies, giving
(P ?x ?y) < (DISTINCT ?x ?y) (R ?y)
Subgoal
(DISTINCT ?x ?y)
Rule (18) applies, giving
(DISTINCT ?x ?y) < (POST (NEQ ?x ?y))
Subgoal
(POST (NEQ ?x ?y))
succeeds binding ?x ← (any ?x1 (NEQ* ?x1 ?y1))
?y ← (any ?y1 (NEQ* ?x1 ?y1))
Proved: (DISTINCT (any ?x1 (NEQ* ?x1 ?y1)) (any ?y1 (NEQ* ?x1 ?y1)))
Subgoal
(R (any ?y1 (NEQ* ?x1 ?y1))
Rule (17) applies
(R B) if we can unify (any ?y1 (NEQ* ?x1 ?y1)) with B
[We try subproof of (NEQ* ?x1 B), which succeeds]

Proved: (P (any ?x1 (NEQ* ?x1 B)) B)

Thus constrained variables may appear in answers. Users may explicitly construct their own constrained variables in queries and assertions as well, if they wish.

Two constrained variables may unify together as long as the combined constraints are provably consistent in a strong sense, i.e., there exists at least one proof of the combined constraints. For example, if we had the following data base:

(23) (PA A)

(24) (PB B)

(25) (PB A)

(26) (T (any ?x (PA ?x)))

We could prove the goal (T (any ?y (PB ?y)) by unification with (26) as follows: (any ?y (PB ?y)) and (any ?x (PA ?x)) may unify to (any ?z (PB ?z) (PA ?z)) if there is an object such that (PB ?z) and (PA ?z). A subproof of (PB ?z) (PA ?z) is found with ?z ← A. This binding is not used, however, since the desired answer could be something else. The result is

(T (any ?z (PA ?z) (PB ?z))).

If in a later part of a proof, ?z was unified against a constant k, a subproof of (PA k) (PB k) would be done before the unification succeeds.

3. Built-In Specialized Reasoning Systems

There are two built-in specialized reasoning systems provided with HORNE. These provide typing for terms and simple equality reasoning.

3.1 Types

All terms in HORNE may be assigned a type. If a term is not explicitly assigned a type, it is assumed to belong in T#C, the universal type. Variables over a type are allowed, and a special syntax is provided. The variable ?x:DOG, for instance, signifies a variable ranging over all objects of type **DOG**. Constants and other ground terms can be asserted to be of a certain type using a built-in predicate ITYPE. Thus

(ITYPE A **DOG**)

asserts that the constant A is of type **DOG**.

Types in HORNE are viewed as sets of objects, and all the normal set relationships between types can be described. Thus one type may be a subset (i.e., subtype) of another, two types may intersect or be disjoint, and the non-null intersection of two types produces a type that is a subtype of the two original types. All this information is asserted using built-in predicates. For example,

(ISUBTYPE **DOG ANIMAL**)

asserts that the type **DOG** is a subset of the type **ANIMAL** (i.e., all dogs are animals).

(DISJOINT **DOG CAT**)

asserts that no object can be both a cat and a dog,

(INTERSECTION FAT-CATS CATS FAT-ANIMALS)

asserts that the set of FAT-CATS consists of all cats that are also fat animals, and

(XSUBTYPE (MALES FEMALES) ANIMALS)

asserts that (MALES FEMALES) is a partition of ANIMALS, i.e., that every animal is either a male or a female, and that all males and females are animals.

All direct consequences of these facts are inferred when the axioms are added. For example, if A and B are disjoint, and A1 is asserted to be a subtype of A, then it is inferred that A1 and B are disjoint. This is done by the forward chaining system. During a proof, the partition information is not used. As a result, asserting (XSUBTYPE (**a b**) **c**) has the same effect as asserting (ISUBTYPE **a c**), (ISUBTYPE **b c**), and (DISJOINT **a b**). During adding type assertions, however, partition information is used. For example, given the relationship between **a**, **b**, and **c** above, if we assert (ISUBTYPE **d c**) and (DISJOINT **d a**), then it will be concluded that (ISUBTYPE **d b**).

The type reasoner acts during unification. A constant will match a variable of type **Tv** only if the constant is of type **Tv** (i.e., the constant is asserted to be of type **Tv**, or is of type **Tvs** which is a subtype of **Tv**). Two variables unify only if the intersection of their types is non-empty. The result is a variable ranging over the intersection of the two types. Thus, complex types may be constructed during a proof. If types **T1** and **T2** intersect, but no name for the intersection is asserted, then a complex type **I(T1 T2)**, which is their intersection, is constructed when unifying ?x:**T1** and ?y:**T2**.

This type reasoner provides a complete reasoning facility between simple types. For complex types, however, the reasoner may permit some intersections that may not be desired since they are empty. Note that this can be checked for at the end of a proof if desired. Any intersection of more than two types is guaranteed only to be pairwise non-empty. For example, if the complex type **I(T1 T2 T3)** is constructed by unifying a variable of type **I(T1 T2)** with a variable of type **T3**, then it must be the case that **I(T1 T2)**, **I(T1 T3)**, and **I(T2 T3)** are non-empty. However, there might be no object that is of type **I(T1 T2 T3)**.

The assertions about the types may be incomplete. For example, two types may be introduced where it is not asserted, or is inferrable, that the types intersect or are disjoint. HORNE provides two modes of proof for dealing with these cases. In the strict mode, two types intersect only if they are known to intersect. In the easy-going mode, two types will intersect unless they are known to be disjoint. Easy-going mode is more expensive, but can be useful in many applications, although it may provide conclusions that on closer inspection are not useful since they contain a variable ranging over the empty set.

As an example, the simple fish data base above could be restated in the typed prover as follows:

(1) (ISUBTYPE COD FISH) <

(2) (ISUBTYPE MACKEREL FISH) <

(3) (ITYPE HOMER COD) <

(4) (ITYPE WILLIE WHALE) <

(5) (LIVE-IN-SEA ?x:FISH) <

(6) (LIVE-IN-SEA ?y:WHALE) <

Although this took one more insertion, it also encodes more information (e.g., whales and fish are disjoint). The proof that WILLIE lives in the sea is much shorter in the typed system. It is completed using only two unifications.

Goal: (LIVE-IN-SEA WILLIE)

unifying with (5) fails as WILLIE is not a fish;

unifying with (6) succeeds, ?y ← WILLIE.

Thus Goal is proved.

If we add the following axioms, we can demonstrate more complicated type reasoning. Let us assume that all animals are either fish or mammals.

(7) (XSUBTYPE (FISH MAMMAL) ANIMALS)

This asserts that both FISH and MAMMAL are subtypes of ANIMAL and that they are disjoint. Note that since COD and MACKEREL are subtypes of FISH, these will also now be disjoint from MAMMALS.

(8) (ISUBTYPE WHALE MAMMAL)

This asserts that WHALE is a subtype of MAMMAL, and hence WHALE is disjoint from FISH.

(9) (ISUBTYPE WHALE THINGS-THAT-SWIM)

(10) (ISUBTYPE FISH THINGS-THAT-SWIM)

Note that in asserting that WHALE is a subtype of THINGS-THAT-SWIM, the system then knows that MAMMAL and THINGS-THAT-SWIM intersect.

(11) (BEAR-LIVE-YOUNG ?m:MAMMAL)

(12) (SWIMS-WELL ?t:THINGS-THAT-SWIM)

Now if we try to find something that bears live young and swims well, i.e., find ?x such that

(BEAR-LIVE-YOUNG ?x) (SWIMS-WELL ?x),

we succeed by unifying the first subgoal to (11), causing ?x ← ?m:MAMMAL, and the second subgoal to (12), causing ?m:MAMMAL and ?t:THINGS-THAT-SWIM to be unified, resulting in a complex variable ?y:I(MAMMAL THINGS-THAT-SWIM). Thus the answer is: all things that are both of type MAMMAL and THINGS-THAT-SWIM. If we add

(13) (LARGE ?w:WHALE)

and query for something that bears live young, swims well, and is large, we will end up unifying ?y:I(MAMMAL THINGS-THAT-SWIM) with ?w:WHALE. The result of this is simply ?w:WHALE, since WHALE is a subtype of both MAMMAL and THINGS-THAT-SWIM.

Constrained variables may be typed in the obvious manner. For example

(any ?x:MAMMAL (SWIMS-WELL ?x:MAMMAL))

is a term that will unify with any term t such that t is of type MAMMAL, and (SWIMS-WELL t) is provable. It is interesting to note that the constrained variable system could be used to implement a typed system directly, where a variable ?x:MAMMAL would be replaced by (any ?x (TYPE ?x MAMMAL)). The semantics of the two notations are identical. Types are so common, however, that the special notation for variables is maintained and types are optimized in the implementation.

Unification between a typed constrained variable and a typed variable results in the expected answers. Thus, unifying ?x:MAMMAL with (any ?y:ANIMAL (SWIMS-WELL ?y:ANIMAL)) succeeds with the result (any ?z:MAMMAL (SWIMS-WELL ?x:MAMMAL)). Unifying ?x:ANIMAL with (any ?y:MAMMAL (SWIMS-WELL ?y:MAMMAL) succeeds simply and ?x:ANIMAL is bound to the constrained variable.

Unifying a constrained variable with a term that itself contains variables may introduce new constrained variables. For example, if we are given the fact (P (f A)), then unifying (any ?x (P ?x)) with (f ?w) will produce the term (f (any ?z (P (f ?z))). This is the correct result since the constrained variable ?x will unify with any term such that (P ?x) is provable. Since (P (f ?z)) is provable (because of the fact (P (f A))), the terms unify. The variable ?w is not bound to A, however, since there may be other terms for which (P (f ?z)) holds as well. Thus (P (f A)) might not be the most general unifier.

These examples are summarized in Figure 1.

Term 1	Term 2	Most General Unifier
(any ?x:MAMMAL (SWIMS-WELL ?x))	WILLIE	WILLIE
(any ?x:MAMMAL (SWIMS-WELL ?x))	?a:ANIMAL	(any ?x:MAMMAL (SWIMS-WELL ?x))
(any ?x:MAMMAL (SWIMS-WELL ?x))	?w:WHALE	(any ?z:WHALE (SWIMS-WELL ?z))
(any ?x (SWIMS-WELL ?x))	(SPOUSE ?a)	(SPOUSE (any ?z (SWIMS-WELL (SPOUSE ?z)))) assuming that the query (SWIMS-WELL (SPOUSE ?a)) succeeds
(any ?x (SWIMS-WELL ?x))	(any ?y (BEAR-LIVE-YOUNG ?y))	(any ?z (SWIMS-WELL ?z) (BEAR-LIVE-YOUNG ?z)) assuming that the query (SWIMS-WELL ?z) (BEAR-LIVE-YOUNG ?z) succeeds

Figure 1: Unification with Constrained Variables

3.2 Typing Functions

Because of the additional complexities involved, a special system is provided for typing functions. This is needed for reasoning about function terms that contain variables. If the only functions used in the system are always fully grounded, the standard type system can be used directly.

For a given function, one can specify the type of the result of the function, plus the types on the arguments of the function. Any function term whose arguments violate these typing restrictions will be flagged as an error. Thus if we define the function SPOUSE to map from PERSON to PERSON, the term (SPOUSE WILLIE) will cause an error, since WILLIE is a WHALE and thus cannot be a PERSON. This function could be defined as follows:

(declare-fn-type 'SPOUSE '(PERSON) 'PERSON),

i.e., the function SPOUSE takes one argument of type PERSON, and produces objects of type PERSON.

Of course, one might like to do better than this, and define SPOUSE to be of type MALE when the argument is FEMALE, and FEMALE when the argument is MALE. Such definitions can be done in HORNE given the following conditions:

1) the function takes a single argument;
2) the function is first declared to the most general type of arguments allowed, and the most general type of objects produced;
3) further declarations are consistent with the other declarations so far;
4) all further declarations have the most general argument type for the specified range type.

In other words,

```
(declare-fn-type 'SPOUSE '(FEMALE) 'MALE)
```

is allowed since

1) it is consistent with the initial definition of spouse;
2) every function with argument type FEMALE produces an instance of type MALE;
3) all function instances of type MALE must have an argument type FEMALE.

Similarly, (declare-fn-type 'SPOUSE '(MALE) 'FEMALE) is allowed.

This will produce the appropriate results during unification. Thus if we unify (SPOUSE ?m:PERSON) with ?x:MALE, the result is (SPOUSE ?m:FEMALE), as desired.

One cannot define a further specification that produces instances of a type already used in a specification, but with a different argument type. For example, the following is not allowed:

```
(declare-fn-type 'fn '(T#U) 'PERSON)
(declare-fn-type 'fn '(MALE) 'MALE)
(declare-fn-type 'fn '(FEMALE) 'MALE) ** ERROR **
```

since the last declaration violates assumption (4) above. Neither MALE nor FEMALE is the most general argument type producing instances of type MALE.

Function typing does not guarantee that functions fully cover their range type (i.e., they are not necessarily "onto"). For example, given

```
(declare-fn-type 'G '(T#U) 'ANIMAL)
```

the query

```
(EQ (G ?x) ?w:WHALE)
```

will *fail*, since there is no guarantee that any terms of form (G ?x) are of type WHALE, even though all are of type ANIMAL. Even if there is a known instance of G of type WHALE, such as (EQ (G ABLE) WILLIE), the above proof will still fail. It is difficult to do otherwise and yet still produce a most general unifier. Some scheme using constrained variables would be possible but would probably be expensive.

3.3 Equality

The system offers full reasoning about equality for ground terms. Thus if you add

(1) (EQ A B) <

(2) (EQ B C) <

(3) (P A) <

you will be able to successfully prove the goal (P B) as well as (P C). Furthermore, given the assertion

(4) (P (f A))

you will be able to successfully prove the goals (P (f B)) and (P (f C)). Adding

(5) (EQ (g A) B)

allows you to prove a potentially infinite class of goals, including (P (g A)), (P (g B)), (P (g C)), (P (g (g A))), (P (g (g B))), etc., to arbitrary depths of nesting of the g function.

An incomplete facility is offered for reasoning about equality for non-ground terms as follows. With a data base of equalities between gounded terms, one can prove an equality statement with variables in it and the variables will be bound appropriately. All possible bindings of the variable are computed and returned in an *any* form so that backtracking to the equality is never needed. Thus if we have

(EQ (f B) G)

(EQ (f A) G)

and we try to prove

(EQ (f ?x) G)

?x will be bound to (any ?x1 (MEMBER* ?x1 (A B))). Multiple variables are also handled correctly by this scheme.

A very limited facility is provided for adding equality statements that contain variables. Essentially, these can be used to prove an equality by a single direct unification. Thus if we add

(EQ (f ?x) (g ?x))

(EQ (f ?x) (h ?x))

we will be able to prove

(EQ (f A) (g A)),

(EQ (f A) (h A)), and

(EQ (f (g ?x)) (h (g ?x)).

but not

(EQ (f A) (h A))

HORNE User's Manual

Version 5.0

September 1984

TABLE OF CONTENTS

1. INTRODUCTION

HORNE is a simple Horn clause theorem prover written in LISP that is a straightforward application of the problem reduction technique to a very simple logical formalism. The original theorem prover, HCPRVR, was written in UCI LISP by Dan Chester at the University of Texas at Austin in August 1979. The software has been completely rewritten by Mark Giuliano and Rich Pelavin to produce compilable LISP code from the axioms, as well as adding many extensions. Questions on using the system should be referred to Mark Giuliano.

HORNE is embedded in a LISP environment. Its facilities are called as LISP functions and HORNE programs can themselves call LISP functions. Thus, effective programming in HORNE involves a careful mixture of logic programming and LISP programming. This manual assumes that the user is familiar with the fundamentals of both LISP and Prolog. The naive user should consult Winston and Horn (1981) for an introduction to LISP, and Kowalski (1974; 1979) and Bowen (1979) for PROLOG. The system is fully implemented, and runs in FRANZ LISP (Foderaro, 1980) on VAX machines.

1.1 Using This Manual

Several notational conventions are followed throughout this manual. *Function* calls that can be made to the HORNE system are shown in italics. HORNE distinguishes between upper and lower case letters. Therefore it is imperative that the reader pay close attention to the case. The usual LISP documentation convention of quoting parameters that are evaluated during function calls is used. For example, in the call

(function-name $\langle arg_1 \rangle$ $\langle arg_2 \rangle$*)*

$\langle arg_2 \rangle$, but not $\langle arg_1 \rangle$, is evaluated. Throughout, all functions ending in the letter "q" do not evaluate their arguments, while most other functions do.

1.2 Syntax

The three major classes of expressions in this language are terms, atomic *formulas*, and *axioms*. The syntax for these classes are given by the following BNF rules:

```
<axiom>              ::= ( <conclusion> ) |
                         ( <conclusion> <index> ) |
                         ( <conclusion> <index> <list of premisses> )
<conclusion>         ::= <atomic formula>
<list of premisses>  ::= <premiss> | <premiss> <list of premisses>
```

```
<premiss>          ::= <variable> | <atomic formula> | /
<index>            := <literal atom> | <list of indexes>
<atomic formula>   ::= ( <predicate name> <list of terms> )
<predicate name>   ::= <constant>
<term>             ::= <constant> | <variable > | ( <list of terms> )
<constant>         ::= <literal atom>
<variable>         ::= ? <literal atom>
<list of terms>    ::= <ε> | <term> | <term> <list of terms> |
                       <term> . <term>
<ε>                ::=
```

An example of an axiom is: ((P ?x) <1 (Q ?x)) where "(P ?x)" is the <conclusion>, "<1" is the index, and "(Q ?x)" is a simple <list of premisses>. This statement is interpreted as follows: the assertion named "<1" signifies that for any x, (Q x) implies (P x). Or, alternately, to prove (P x) for any x, try to prove (Q x).

1.3 Special Symbols

The HORNE system uses two special symbols which should not be used for other purposes:

> "?" indicates a variable will cause the atom following it to be expanded into the internal variable format. This is true only in axioms. The symbol can be used freely in LISP code.
>
> " = " is a read macro that is used for HORNE comments, and so should be avoided entirely. If comments are not desired, this can be disabled using the appropriate LISP reader functions.

2. BASIC HORNE PROGRAMMING

This section explains how the HORNE database can be modified and examined. and how theorems can be proved.

2.1 Defining and Deleting Predicates

Several simple functions are available for asserting and retracting axioms.

(axioms '<list of axioms>)

> Asserts all of the axioms in *<list of axioms>* at the end of the database in the order they appear in the list. Same as *addz*.

(adda '<axiom$_1$> ... '<axiom$_n$>) and *(addaq <axiom$_1$> ... <axiom$_n$>)*

Adds all the axioms to the beginning of the database. *<axiom1>* will precede *<axiom2>* in the database, etc. Warning: This operation is much more expensive than *addz* or *axioms*.

(addz '<axiom$_1$> ... '<axiom$_n$>) and *(addzq <axiom$_1$> ... <axiom$_n$>)*

Adds all the axioms to the end of the database. *<axiom1>* will precede *<axiom2>* in the database.

(retracta '<predicate name>) and *(retractaq <predicate name>)*

Retracts the first axiom in the database that concerns *<predicate name>*.

(retractz '<predicate name>) and *(retractzq <predicate name>)*

Retracts the last axiom in the database that concerns *<predicate name>*.

(retractall '<pattern>) and *(retractallq <pattern>)*

Retracts all the axioms in the database whose conclusions unify with the specified pattern. The predicate name must be specified in the pattern. If an atom is given as a pattern, it will be interpreted as a predicate name and all axioms for that predicate will be deleted. For example, (rall '(P A ?x)) retracts all axioms whose head unifies with (P A ?x) (e.g., (P ?x ?z), (P ?x B), (P A B)), and (rall 'P) retracts all axioms for predicate P.

(clear '<index>) and *(clearq <index>)*

Retracts all axioms in the database with an index matching the specific index. This function accepts patterns for complex indexes. Thus (clear '(ff ?x)) would delete all axioms with an index consisting of a two-element list with the first atom being "ff" (e.g., (ff 1), (ff DD), (ff (aa b))).

(clearall)

Deletes all axioms defined by the user.

Predicates in HORNE can either have a constant arity or can vary. The addition mechanism assumes that any predicate not previously specified as a varying predicate is constant. To define a predicate with a varying number of arguments, use the function

(declare-varyingq <predname1> ... <prednamen>),

e.g.,

(declare-varyingq or* and*)

The predicate or* defined in Section 5.3 is an example of a predicate that has to be declared to be varying. Only varying predicates allow list matching on their arguments. Thus, for or*, we can use a term of form (or* ?first . ?rest) and the variables will be matched appropriately.

2.2 Examining the Database

The database of axioms can be examined with the following functions:

(printp '<pattern>) and *(printpq <pattern>)*

Pretty prints all of the axioms whose conclusions unify with the pattern, including comments. As with *rall*, atomic patterns are assumed to be predicate names.

(printi '<index>) and *(printiq <index>)*

Pretty prints all of the axioms that have an index that unifies with the specified index.

(relations)

Returns a list of all the predicate names currently defined in the system. This includes all of the predicate names that are LISP functions.

(indices)

Returns a list of all the indices in use.

(axioms-by-index '<index>)

Returns a list of axiom names associated with the given index. This uses a direct match of the index without unification.

(axioms-by-name-and-index '<pred-name> '<index>)

Prints all the axioms with the given predicate name and the given index. This uses a direct match of the index without unification.

There are also functions for accessing the data base without invoking the prover:

(find-facts '<atomic formula>) and *(find-factsq <atomic formula>)*

Returns all axioms of form *(<conclusion>)* or *(<conclusion> <index>)* that unify with the specified formula. Thus to find all axioms that assert that P is true of something, we could use (find-facts '(P ?x)). If the data base contained the facts

((P A))
((P B) <3)
((P D) <4 (Q R))

then the query would return (((P B) <3) ((P A))).

(find-facts-with-bindings '<atomic formula>)

Same as *find-facts* except that it returns the variable bindings as well in the format ((<axiom> <binding list>)*). For example, with the above three axioms for P, the query (find-facts-with-bindings '(P ?x)) would return

((((P B) <3) ((?x B))) (((P A)) ((?x A)))).

(find-clauses '<atomic formula>)

Returns all axioms whose *<conclusion>* unifies with the specified formula. The same restrictions on variable naming as with *find-fact* hold for this function. It would return all three of the above axioms in the query (find-clause '(P ?x).

(get-facts '<atomic formula>)

Same as *find-facts* except that the <conclusion> must be identical to the specified formula ignoring variable naming, e.g., (get-facts '(P ?x)) with the above three axioms would return NIL.

(get-clauses '<atomic formula>)

Same as *find-clauses* except that the <conclusion> must be identical to the specified formula ignoring variable naming.

2.3 Proving Theorems

The theorem prover is invoked by calling the LISP function *prove* with a set of formulas that represent the goal clause.

(prove '<atomic formula$_1$> ... '<atomic formula$_n$>)
(proveq <atomic formula$_1$> ... <atomic formula$_n$>)

Attempts to prove the list of formulas, and returns a bound solution if one is found.

Once a proof is completed, you can find out the execution time in seconds by calling (*runtime*). The answer returned by the last query can be printed using the function (*printanswer*).

There are variations on the *prove* command that allow multiple answers to be found. These are indicated by an optional first argument as follows:

(prove query '<atomic formula$_1$> ... '<atomic formula$_n$>)
(proveq query <atomic formula$_1$> ... <atomic formula$_n$>)

Prompts the user each time a solution is found, and queries whether to search for another or not.

(prove all '<atomic formula$_1$> ... '<atomic formula$_n$>)
(proveq all <atomic formula$_1$> ... <atomic formula$_n$>)

Does an entire search of the axioms and returns all solutions found. Note that currently if there is an infinite path in the proof tree (e.g., a transitivity axiom) then this function will not return.

(prove <number> '<atomic formula$_1$> ... '<atomic formula$_n$>)
(proveq <number> <atomic formula$_1$> ... <atomic formula$_n$>)

Finds *<number>* proofs of the goal obtained by evaluating '<formula>. Note that *(prove 1 <formula>)* is equivalent to *(prove '<formula>)*.

Note: Every 500 proof steps the theorem prover prompts the user whether to continue or not. When you see the output "*continue?*", respond with a "y" to continue, "n" to stop. Also at this point, any LISP function can be evaluated and the system will then reprompt whether to continue. See Section 7 to change the number of steps before a prompt.

2.4 Comments

Comments can be added for each predicate name. These are then printed by the various print functions.

(add-comment '<predname> '<comment>)

Adds a comment to the predicate specified (and deletes any existing comment). The comment can be any LISP expression, but it is most convenient to use strings, e.g.,

(add-comment 'loves '|This is a comment|)

Strings can include carriage returns, so longer comments can be used.

(add-to-comment '<predname> '<comment>)

Extends an existing set of comments with the new comment.

(print-comment '<predname>)

Prints the comments for a predicate.

3. THE PREDICATE EDITOR

The axioms of a single predicate can be defined and modified using the HORNE predicate editor, which is entered with the function *(edita <predicate name>)*. An online help facility is provided with the editor. Once the editor has been entered, the following commands are available:

p Print the axioms with numbers.

q (Quit) Complete the edit.

u Undo all changes made to the axioms (i.e., complete restart).

a <number>

Add an axiom at indicated position. You will be prompted for the axiom. If index is "z" then axiom is added at the end.

r <number1> ... <numbern>

Delete the indicated axioms. The remainder axioms are renumbered.

e Enter intra-axiom editor mode. Single axioms may be edited using the LISP editor in this mode. On entering this mode you will be prompted for the number of the axiom to be edited. See the LISP manual for details on the LISP editor. Warning: The LISP editor should only be invoked through this *edita* function, simply calling *editp* on the predicate name will produce disasterous results as the axioms share common sublists.

m <number1> ... <number2>

Move axiom number <number1> to position <number2>.

4. TRACING AND DEBUGGING IN HORNE

The HORNE system provides extensive tracing facilities that operate on the entire proof, or on selected predicates. There are four places where tracing may occur during the processing of a single goal. These are called the *q*, *a*, *b*, and *r* tracepoints throughout, and are defined as follows:

- The *q* tracepoint is the point where the goal is first selected by the prover;
- the *a* tracepoint is the point where a clause is selected in an attempt to prove the goal:
- the *b* tracepoint is the point where the prover resumes after backtracking (note that the *b* points are a proper subset of the *a* points):
- the *r* tracepoint is the point where the goal has been proven and the prover is "returning" to consider a new goal.

In every trace function you can explicitly specify which tracepoints you want. If they are not specified, the default is the *q* and *r* tracepoints.

4.1 Global Tracing Controls

(htraceall)

When called it turns on a trace of HORNE showing every formula that is about to be proved (i.e., at the *q* tracepoint), as well as indicating when a formula has been proved (i.e., at the *r* tracepoint). It can take the following optional specifications:

(at <tracepoint>)

Indicates tracing at the specified tracepoints only, e.g., (htraceall (at q b)) traces all predicates at the query and backtracking points.

break

Indicates a break is desired in addition to a trace message. See 4.3 for a description of the break package.

(using <LISP function>)

Indicates that a user-supplied function should be called at the tracepoint rather than printing a message. See Section 4.4 for details.

These can be combined as you wish. For instance, if you want a break at backtracking points, and a trace of query points, use

(htraceall break (at b))

(htraceall (at q))

4.2 Selective Tracing

The user can trace individual goals by identifying which predicate names are to be traced. The simple form of this function is described first, then further options are introduced.

(htrace <predspec$_1$> ... <predspec$_n$>) or
(htraceq <predspec$_1$> ... <predspec$_n$>)

When <predspec> is a simple predicate name (e.g., (htraceq P)), this causes tracing at the *q* and *r* tracepoints of all goals that have the specified predicate name as their head. When <predspec> is a list of form (<predname> <options>*), the user can specify various options as described in Section 4.1. For example, *(htraceq (P (at q all))* traces P at the tracepoints q and a.

(unhtrace <predicate name$_1$> ... <predicate name$_n$>) or
(unhtraceq <predicate name$_1$> ... <predicate name$_2$>)

Turns off selective tracing. If no predicates are specified, all selective tracing is undone.

A similar set of tracing facilities are provided for tracing by the index of clauses rather than the predicate name in the conclusion. In index tracing, however, only the *a* and *b* tracepoints can be specified.

(htraceiq <index-spec$_1$> ... <index-spec$_n$>)

Turns on tracing for the specified index.

An <index-spec> is of the following form:

(<index pattern> <options>)*

An <index pattern> is an expression that may contain HORNE variables. Any clause with one index that unifies with the pattern is traced. For example, *(rtraceiq (<1) (<3))* would cause tracing at all *a* tracepoints that use a clause with index "<1" or "<3," and *(rtraceiq ((<G ?x)) ((F ?x) break))* would cause tracing at all *a* tracepoints using a clause with an index unifying with (<G ?x), and cause a break at all *a* tracepoints using a clause with an index unifying with (F ?x).

(unhtraceiq <index$_1$> ... <index$_n$>)

Undoes the above trace commands. If these are called with no arguments, all index tracing is turned off.

The trace messages all involve printing out formulas. To control the I/O behavior one can set limits on how deep a formula will be printed, as well as the length. This is controlled by the global variables:

H$$DEPTH - the depth to which formulas will be printed (default is 4).
H$$LENGTH - the length of formulas to be printed (default is 6).

4.3 The Break Package and Traces of Proofs

Once a proof is interrupted using a break in the trace package, the programmer can look around at what is happening, modify the tracing behavior, etc. To continue the proof, enter *go*. Some useful functions for debugging are:

(goal)--prints the current formula to be proved.

(top)--prints the current top of the goal stack.

(stack)--prints the current goal stack (see below).

(show-proof-trace)--prints a trace of the proof up to the current point (see below).

(show-facts)--prints the axioms that could directly prove the goal.

(show-clauses)--prints the clauses that could be used to prove the goal.

(totry)--prints all axioms for current predicate name.

The goal stack contains the current formula being proved at each level of recursion, plus all the succeeding formulas that need to be proven once the current formula succeeds. Thus if we had the axioms

((A) < (B) (C) (D))
((B) <)
((C) < (E) (F))

and we put a break on the predicate in E (i.e., (rtraceq (E break)), in trying to prove A we would find the following stack at the break point:

((E) (F))
((C) (D)).

In other words, we're trying to prove E, after which we will try to prove F. If both succeed then we will have proven C, and will try to prove D.

Any valid LISP expression can also be evaluated while debugging.

After a proof has been found, one can obtain a full trace of the successful proof tree. If multiple proofs are found, a list containing each individual proof is returned. For efficiency reasons, however, a proof trace is not collected unless some predicate is being traced. If you wish a proof trace to be constructed when nothing is being traced, you must first call *(turn-on-proof-trace)*. The function *(turn-off-proof-trace)* puts the system back into its default mode.

(proof-trace)

> Returns the successful proof tree(s) of the last call to the prover, or, if called within a proof break, returns the current state of the proof tree. For formatted printing of the trace, you can call *(show-proof-trace)*.

The format of the proof tree is *(<conclusion> <index> <proof-trace of subgoals>)*.

Thus, given the axioms

```
(A <1 B C)
(B <2 D)
(C <3)
(D <4)
```

if we proved the goal A, the proof tree would be

```
(A <1 (B <2 (D <4))
      (C <3))
```

4.4 User Defined Trace Functions

Users can define their own tracing functions for use in the HORNE system. All tracing functions must have the same form: they must be lambda expressions taking two arguments. The first is set to the type of tracepoint (i.e., either *q*, *a*, *b*, or *r*) and the second is the instantiated clause that caused the trace. The default tracer simply prints this information at the terminal after some formatting. For example, we could define our own trace function as follows:

```
(def ttt
   (lambda (tpoint clause)
      (terpri)
      (print (list tpoint clause)))
```

Then given the three axioms:

(P ?x) < (Q ?x ?y) (R ?y)
(Q A ?z)
(R B)

and the trace command

(traceall (using ttt)),

we get the following output during the proof of (P ?d):

(q (P ?d))
(q (Q ?d ?y1))
(r (Q A ?y1))
(q (R ?y1))
(r (R B))
(r (P A))

5. THE HORNE/LISP INTERFACE

So far, we have seen how the various HORNE facilities can be invoked from within LISP. This section explains how LISP facilities can be used within HORNE.

5.1 Assigning LISP Values to HORNE Variables

There is a simple mechanism for binding a HORNE variable to an arbitrary LISP value. This is accomplished by using the built-in predicate:

(SETVALUE <variable> <LISP expression>)

> This evaluates the <LISP expression> as a LISP program and binds the result to the HORNE variable specified. If the variable is already bounded, SETVALUE will fail.

(GENVALUE <variable> <LISP expression>)

> This is the same as SETVALUE except that the LISP expression is expected to return a list of values. The variable will be bound to the first value, and if the proof backtracks to this point, to the succeeding values one at a time.

5.2 Predicate Names as LISP Functions

Occasionally it is useful to let a predicate name be a LISP function that gets called instead of letting HORNE prove the formula as usual. The predicate name "NEQ", for example, tests its two arguments for inequality by means of a LISP function because it would be impractical to have axioms of the form ((NEQ X Y)) for every pair of constants X and Y. These special LISP functions must be FEXPRs and receive their argument list from HORNE with all bound variables replaced by their values. To declare such a LISP function to HORNE use

> *(declare-lispfnq <name$_1$> ... <name$_n$>)*

From then on HORNE will recognize those <name>s as LISP functions. LISP functions should only return "t" or "nil" which will be interpreted as true and false respectively. For example, assume we enter the following:

```
(def check
     (nlambda (x)
          (terpri)
          (princ '|in check, args are: |)
          (print x)))
(declare-lispfnq check)
(addzq ((P ?x ?y) < (check ?x ?y)))
```

Then if we call

(proveq (P A B))

the LISP function check is called resulting in the output:

in check, args are: (A B).

Since check returns a non nil answer, the LISP call is treated as a success.

Other useful functions for manipulating argument lists within LISP are:

(*isvariable* '*<term>*)

Returns the variable name if <term> is an unbound HORNE variable; otherwise it returns nil.

(*vartype* '*<variable>*)

Returns the type of the HORNE variable, or nil otherwise.

(*bind* '*<variable>* '*<value>*)

Binds the HORNE variable to the value of the LISP expression. If the first argument is not a HORNE variable, it returns nil. Example: the following LISP function sets the first HORNE argument to 4 if it is a variable:

```
(def SetTo4
   (nlambda (x)
      (cond ((isvariable (car x))
             (bind (car x) (add1 3))))))
```

5.3 Using Lists in HORNE

Since HORNE is embedded in LISP, one can use the LISP list facility directly. In fact, the HORNE unifier can be thought of both as operating on logical formulas, and matching arbitrary list structures.

The unifier will handle the dot operator appropriately anywhere except at the top level of non-varying predicates. Thus the following pairs of terms unify with the most general unifier shown:

(a b c)	(a ?x ?y)	with m.g.u. {?x/b, ?y/c}
(a b c)	(a . ?x)	with m.g.u. {?x/(b c)}
(a b c)	(?x . ?y)	with m.g.u. {?x/a, ?y/(b c)}
(a b c)	(a ?x . ?y)	with m.g.u. {?x/b, ?y/(c)}
(a b)	(a ?x .?y)	with m.g.u. {?x/b ?y/nil}
(a)	(a ?x .?y)	does not unify.
(a b)	(?x)	does not unify. (?x) only matches lists of length 1.

List unification is also allowed with varying arity predicates, although the predicate name position cannot contain a variable. Consider the definition of the predicate *or** that is true if any of its arguments is true:

(declare-varyingq or)*
((or ?x .?y) < ?x)* or* is true if the first argument is true
((or ?x . ?y) < (or* . ?y))* or* is true if or* of all but the first argument is true

Thus the call with no arguments, *(or*)*, always fails and each of *(or* (A))*, *(or* (B) (A))* and *(or* (B) (A) (C))* succeeds if *(A)* is provable.

5.4 Manipulating Answers from HORNE

Once a proof succeeds, these commands can manipulate the answer returned.

(get-binding '<varname>)

Returns the binding for the named variable. For example, *(get-binding '?x)* will return the binding for ?x in the last proof. If multiple solutions were found in the last proof, a list of bindings is returned.

(get-answer)

Returns the answer found in the last query. If multiple answers are found, a list of answers is returned.

6. SAVING AND RESTORING PROGRAMS

These commands allow the user to partially or entirely save his HORNE program and to restore it at a later time. Great difficulty can be saved by avoiding the use of "." in filenames.

(get-axioms '<filename>) and *(get-axiomsq <filename>)*

Retrieves the axioms and LISP predicates that have been saved in *<filename>* by save-axioms. The names of the predicates defined by this retrieval are put in a list named (concat <filename> 'fns). Thus *(get-axioms xxx)* reads in the predicates in file xxx, and sets the variable xxxfns to the names of the predicates that were restored from xxx.

(save-predicates '<filename> '<list of prednames>)

Saves the axioms and comments for the predicates given in the specified file. LISP predicates declared to HORNE may also be saved. The output is in a pretty format (with "?" for variables). Hashtable info is saved so they can be reconstructed when retrieved. If <filename> already exists, a backup copy is retained as <filename>.back.

(save-all-axioms '<filename>)

Does a save-predicates on all the predicates known to the system.

(save-indices '<filename> '<list of indices>)

Saves all axioms with one of the specified indices on the specified file. The output is in pretty format, but no comments are saved. No hashtable info is saved.

(dump-predicates '<filename> '<list of prednames>)

This saves the definitions of the predicates specified in the file in an internal format. Thus reading in the file is considerably faster, but the file is not for human consumption. If the second argument is omitted, all the known predicates are dumped. *Dump-predicates* always saves all the type information even if only a subset of the defined predicates are dumped. Dumped files are compilable by the LISP compiler, whose output can then be loaded into HORNE.

7. TYPED THEOREM PROVING

The type of a variable is indicated by following the variable with the name of its type. Thus *?x:CAT* names a variable *?x* that is of type CAT. The variable *?x:CAT* will unify only with terms that are compatible with the type CAT. The internal format for typed variables is the list *(* # <name> . <type>)* as in *(* 3 ?x . CAT)*.

Types should be viewed as sets, and no restrictions are assumed as to whether sets are disjoint, mutually exclusive, or wholly contained by each other. This information is specified by the user with assertions of the forms:

(ITYPE <individual> <typename>)

Asserts that the individual is of the indicated type, e.g., (TYPE A CAT) asserts that the constant A is of type CAT.

(ISUBTYPE <subtype> <supertype>)

Asserts that the first type is a subclass of the second type, e.g., (SUBTYPE CAT ANIMAL) asserts that CAT is a subclass of ANIMAL.

(DISJOINT <type1> <type2> ... <type n>)

Asserts that all the types mentioned are pairwise disjoint.

(INTERSECTION <newtype> <type1> <type2>)

Asserts that the intersection of type1 and type2 is newtype.

(XSUBTYPE (<type1> <type2> ... <type n>) <super-type>)

Asserts that type1 ... type n is a partition of sup-type, i.e., they are all subtypes of sup-type. that type1 ... type n are pairwise disjoint, and that the union of type1 ... type n is equivalent to sup-type.

7.1 Adding TYPE Axioms

These statements are added to HORNE in the form of axioms by using the regular axiom addition functions *adda*, *addz*. *axioms*. etc. However. two things occur when axioms of these forms are added:

1) The relation between the types named and its implications are added to a matrix which stores the known set relationship between all the types known to the system. Of course what is implied by any statement depends on what is already in the matrix.
2) The statement is added to the axiom list so they can be printed out and edited as normal axioms.

The system that adds a TYPE axiom and its implications to the matrix first checks that the statement is consistent. If the statement contains an inconsistency, an error message is printed and no information is added to the matrix. For example, if one adds (DISJOINT cats dogs) and then adds (SUBTYPE dogs cats), an error message will be given and information in the second axiom will not be added to the matrix.

In order for the matrix system to derive all implied information, ITYPE axioms should be added after SUBTYPE. XSUBTYPE. DISJOINT, and INTERSECTION axioms. Adding an ITYPE axiom may add or delete other ITYPE axioms implied by the axiom. (In fact, sometimes the axiom that was written might not even be added.) Because of this and the nature of axiom addition, axioms for the predicate ITYPE are always added at the end of the axiom list for ITYPE (e.g., as with using *addz*). This restriction has no effect on the proof procedure, for the order of the atomic ITYPE axioms is irrelevant. *Edita* can be used to reorder the axioms for documentation purposes.

Type restrictions on the arguments to a function term, and on the type of the function term itself, are declared using the form:

(declare-fn-type '<fn-name> '(<type1> ... <typen>) '<typename>)
(declare-fn-typeq <fn-name> (<type1> ... <typen>) <typename>)

Asserts that <fn-name> is the name of a function that takes arguments of the types <type1>,...,<typen> and describes objects of type <typename>. For example, (declare-fn-type ADD (NUMBER NUMBER) NUMBER) declares a two-place function ADD, with both arguments of type NUMBER, and which produces an object of type NUMBER.

Single place functions may have multiple declarations subject to strict conditions outlined below:

1) the first declaration is the most general in its argument place and its value;
2) all subsequent declarations define a proper subset of the first definition in both the argument type and the value type;
3) the type of the argument is the most general type that produces values of the specified value type.

Examples and further discussion are found in the system overview, Section 3.2.

Declare-fn-type returns one of three values to indicate the status of the call:

t -- a new definition of a type (or exact repeat of a previous definition)

comp -- an additional definition to a single argument function that is compatible with all previous definitions

nil -- improper form of definition or a definition inconsistent with previous definitions

(delete-fn-definition '<function name>)

Removes all previous definitions for the function.

7.2 Deleting TYPE Axioms

In order to delete an axiom about types, one can use one of the HORNE deletion functions (retracta, retractz, edita, retractall, etc.). However, at this point, the prover is disabled. This is because the axiom lists are correct but the matrix has not been changed. In order to restore the matrix and enable the prover to run, use the function:

(recompile-matrix)

Recompiles all the type axioms in the system.

This is an expensive process and should be avoided if possible.

7.3 LISP Interface to Type System

There is a set of LISP functions to access and use the type system independently of HORNE. The most important function returns the type of an arbitrary HORNE term:

(get-type-object '<term>)

Given any HORNE term, this function returns the most specific type of that term. If the term contains one or more variables, it returns the most specific type that includes every instantiation of the term.

(issub '<type1> '<type2>)

Takes any two types and returns t if the types are identical, or if <type1> is a proper subtype of <type2>.

There are functions for inspecting the definitions of function terms (in addition to get-type-object above).

(see-function-definition '<function name>)

Returns the complete type table for the specified function. For single argument function, this may be a tree of the form

(<function type> (<arg type list>) <subtree>*).

For example, the function SPOUSE might have the definition

(PERSON (PERSON) (FEMALE (MALE)) (MALE (FEMALE)))

i.e., SPOUSE of a PERSON is of type PERSON, and SPOUSE of MALE is of type FEMALE, and SPOUSE of FEMALE is of type MALE.

(defined-functions)

Returns a list of all function names that have been declared.

One can examine the TYPE axioms added to the system by using the HORNE functions *printp*, *printi*, etc., but these functions will only show you the base facts and not all the inferences the system has made. The following functions allow examination of what is in the matrix.

(matrix-relation '<type1> '<type2>)

Returns the information that is stored in the matrix for the relationship between the two types.

(type-info '<type>)

Returns a list giving the relationship between the given type and every other type in the system, of the form: ((type rel type1)(type2 rel type) ...) The type you are querying can be in either the first or second slot.

The following are the possible relationships between types:

1) "sb"--a subset relation holds between the two types.
2) "ss"--a superset relation holds.
3) "o"--the types intersect but the overlap is not named.
4) "(ip (list))" or "(p (list))"--a superset partition relationship holds: the list contains all the partitioning sets of the superset.
5) a list of length 1--the item on the list is the name of the intersection of the given types.

(types)

Returns a list of all types known in the system.

7.4 Type Compatibility and An Example

Using the axioms above, HORNE can compute the compatibility of two terms efficiently. Types are compatible if one is a subtype of the other or if they overlap. Overlaps occur in two ways: named or unnamed. A named overlap results from an INTERSECTION axiom; an unnamed overlap can be implied from either TYPE axioms or a named overlap. The unification of two typed variables may result in a variable of a complex type of the form *(int type1 type2)* indicating the intersection of the two types. This new type is recognized in the proof as a new type. For example, suppose we have the axioms:

(ISUBTYPE cars anything)
(ISUBTYPE person anything)
(ISUBTYPE ford cars)

```
(ISUBTYPE smallcars cars)
(ISUBTYPE student person)
(ISUBTYPE worker person)
(ITYPE john worker)
(ITYPE john student)              ; note this implies that the types worker
(INTERSECTION pintos ford smallcars)          and student overlap
((want ?x:person ?g:ford) < (fuel-efficient ?g:ford)
                            (wealthy ?x:person))
((fuel-efficient ?f:smallcars) <)
((wealthy ?d:worker) <)
```

We could then query (want ?f:student ?d:ford) and we would get (want ?r:(int student worker) ?u:pintos), pintos being a named overlap while the intersection of the types *student* and *worker* is derived by the prover.

7.5 Tracing Typechecking

In order to trace the typechecking functions, call the function (*trace-typechecking*). The prover will break during typechecking if this function is called with the form (*trace-typechecking break*). In order to stop tracing, call (*untrace-typechecking*).

7.6 Assumption Mode

The default mode for HORNE is to assume that two types whose relationship is not known are not compatible. This can be overridden by the command (*type-assumption-mode*), in which all unknown relationships are assumed to be unnamed intersections. Alternatively, the mode (*type-query-mode*) will query the user each time two types are found for which there is no known relationship. The function (*normal-type-mode*) returns the system to default mode.

In assumption mode, the format of answers is

((<answer> <type assumptions>)).

For example, given (Q ?x:CAT) and proving (Q ?x:DOG) in assumption mode where no relationship is known between the types CAT and DOG, we get:

(((Q ?x:(int CAT DOG)) (int CAT DOG)))

Note that if you obtain multiple answers in this mode, the list of assumptions for each answer may refer to assumptions needed for other answers as well.

7.7 Defining a Custom Typechecker

If users wish to design their own type checking facility, the interface between the unifier and the type checking system consists of two LISP functions that can be rewritten. These are:

(typecheck <term> <type>)

Returns *t* if and only if the term is of the appropriate type (or a subtype);

(typecompat <type1> <type2>)

Returns the more specific type. For example.

(typecompat GIRL PEOPLE) returns *GIRL*,
(typecompat GIRL BOY) returns *nil*.

8. EXTENSIONS TO THE UNIFICATION ALGORITHM

The unifier in HORNE has been augmented to allow two types of special unification dealing with equality and restricted variables.

8.1 Equality

The unification algorithm of HORNE has been modified so that when terms do not unify they can be matched by proving that the terms are equal. Any variables in the terms matched will be bound as needed to establish the equality. Equality statements are added to the system by using the axiom EQ. (Note that EQ is of arity 2.) For example:

((EQ (president USA) Ronald-Reagan) <)

expresses a fact that is well known to most Americans. The axiom

(EQ (add-zero 1) 1) <)

expresses an infinite class of equalities. For example, (add-zero (add-zero 1)) equals 1, as does (add-zero (add-zero (add-zero 1))), and so on.

The system provides, in an efficient manner, complete reasoning about fully grounded terms (i.e., terms that contain no variables), and supports partial reasoning about equality assertions containing variables. The current system will allow variables in queries (which may be bound to establish equalities), but variables in equality assertions are restricted in their use. In particular, there is no transitivity reasoning for terms containing variables; e.g., given

(EQ (f ?x) ?x)
(EQ (G ?y) (f ?y))

we can prove (EQ (f A) A), (EQ (f ?z) ?z), and (EQ (G (f ?t)) (f ?t)), but cannot prove (EQ (G A) A), even though it is a logical consequence of the two axioms above.

The information derived from the EQ axioms that are asserted is stored on a pre-computed table which is updated as EQ axioms are added and deleted. This table is storable using the standard HORNE I/O mechanism.

There are two LISP functions for examining the equality assertions:

(equivclass <ground term>)

Returns a list of all ground terms equal to the <ground term>.

(equivclass-v <term>)

Returns a list of all terms that could be equal to the term followed by variable binding information.

8.2 The Post-Constraint Mechanism

HORNE allows the user to specify that the proof of an atomic formula be delayed until the terms in it are completely bound. The user does this by enclosing the atomic formula within the lispfn POST, as in the axiom:

((F ?x) < (POST (MEMBER ?x (a very very long list))) (G ?x)).

POST takes an atomic formula as an argument. If the formula is grounded then the proof proceeds as usual. Otherwise the variables in the formula are bound to a function which restricts its value and the proof proceeds as though the proof of the formula succeeded.

Restrictions on variables are implemented by binding the variable to a special form

(any ?newvar (constraint ?newvar)).

Thus, give the above axiom, if we queried (F ?s) the POST mechanism would bind ?s to

(any ?s0001 (MEMBER ?s0001 (a very very long list))).

This use of a special form *any* is similar to the *omega* form used in Kornfeld (1983).

The HORNE unifier has been modified so that it knows about *any*. A term of form (any ?x (R ?x)) will unify with any term that satisfies the constraint (R ?x). Again using the above axiom: after the POST succeeds, the proof continues with the subgoal

(G (any ?s0001 (MEMBER ?s0001 (a very ....].

Now suppose that (G e) is true. Then we can unify these two literals if we can prove

(MEMBER e (a very very long list)).

Note that the constraint will be queried only once its variable is bound. Thus if (G ?c) were true above, the unification would succeed and

(F (any ?s0001 (MEMBER ?s0001 (a very long list))))

would be returned as the result of the proof. If (G (fn ?c)) were true instead, a recursive proof testing whether (MEMBER (fn ?c) (a very very long list)) would be done and, if successful, the final result of the proof would be

(F (fn (any ?z (MEMBER (fn ?z) (a very very long list))))).

During normal tracing, any subproofs due to the post constraint mechanism are not traced. If tracing is desired for these proofs, call *(htrace-post-proof)*. To set it back to the default of no tracing, call *(unhtrace-post-proof)*.

8.3 Interaction Between Systems

The equality system and the POST mechanism use each other as can be shown by the following example.

(EQ (child-of Adam) Abel)
(EQ (child-of Eve) Abel

Then we can unify (child-of ?x) with Abel, resulting in ?x being bound to

(any ?x0001 (MEMBER ?x0001 (Adam, Eve))).

Thus we have restricted the values that ?x can take on to Adam or Eve. It should be noted that MEMBER must take equality into account; that is, in the example, the *any* term should unify with the term (First-man) given (EQ (First-man Adam)).

9. THE FORWARD CHAINING FACILITY

The prover has a forward production system in which the addition of new axioms adds new facts that are implied by the existing axioms. The general form of forward axioms are as follows:

((trigger) (list of conclusions) index (list of conditions)).

After a HORNE axiom is added to the database it is checked to see if it matches any trigger pattern. A trigger must be an atomic formula, but cannot be a LISP predicate. If it matches, then using the binding list of the match the system tries to show that the conditions associated with the trigger are in the database. Note that the system does not try to prove the conditions (unless specified), but simply checks that they are in the database. If all the conditions can be shown to be in the database then each of the conclusions in the conclusion list is added to the HORNE axiom list using the bindings collected in the process. LISP predicates can be used in the conditions and in the conclusions, where they are called as in the backwards chaining system. The value returned by a LISP predicate in the conclusion list is ignored. In adding a conclusion another trigger may be fired. To prevent infinite looping the forward chaining system will not add axioms that are already in the database.

9.1 Defining Forward Production Axioms

(addf '<atomic formula> '(<atomic formula> ...) '<index>
'(<atomic formula> ...))
(addfq <atomic formula> (<atomic formula> ...) <index> (<atomic formula> ...))

Adds the forward production axiom to the end of the data base, e.g., adding the following

```
(addf '(e ?d) '((w ?d)) 'r '((r ?d))))
(addaq ((r d) s))
(addaq ((e ?f) j))
```

will result in the axiom ((w d) r) being added to the database.

9.1.1 Options to addf and addfq

(addf 'all '(<atomic formula> ...) '<index> '(<atomic formula> ...))
(addfq all (<atomic formula) ...) <index> (<atomic formula> ...))

Using the atom "all" for the trigger adds a separate forward-chaining axiom for each of the atomic formulas in the condition list with that condition as the trigger. Thus each of the conditions is a trigger, e.g.,

```
(addf 'all '(eq ?y ?z) 'eq '((eq ?y ?x) (eq ?x ?z)))
```

adds the following axioms to the system:

1. (eq ?y ?x) (eq ?y ?z) eq ((eq ?y ?x) (eq ?x ?z))
2. (eq ?x ?z) (eq ?y ?z) eq ((eq ?y ?x) (eq ?x ?z))

Upon the following addition:

(addaq ((eq w e) l)
(addaq ((eq r w) l)

the axiom ((eq r e) eq) is added to the system.

(addf '<atomic formula> '(<atomic formula> ...) '<index> '())
(addfq <atomic formula> (<atomic formula> ...) <index> ())

Using "()" for the conditions list makes it such that whenever the axiom is triggered it will assert its conclusions.

(addf '<atomic formula> '(<atomic formula> ...) '<index>
'(<atomic formula>) ...)
(addfq <atomic formula> (<atomic formula> ...) <index>
(<atomic formula>) ...)

This option allows a lispfn to occupy the position of the predicate name in any of the conditions. The lispfn succeeds if it returns a non nil value.

(addf '<atomic formula> '(<atomic formula> ...) '<index>
'((prove <atomic formula>) ...))
(addf <atomic formula> (<atomic formula> ...) <index>
((prove <atomic formula>) ...))

The prove option allows any of the conditions to call the theorem prover to prove the condidtion. (Note that normally conditions are not proved but just shown to be in the data base). The condition is true if the atomic formula can proved by the theorem prover. Any variables bound in the proof will be passed on to the next condition.

(retract-forward 'form) and (retract-forwardq form)

These delete the forward-chaining axioms specified by the given form, which is either a pattern or a predicate name. If the form is a predicate name, all forward-chaining axioms that have the given predicate name as their trigger name are deleted. Otherwise all forward-chaining axioms whose trigger unifies with the given pattern are deleted. Note that if the form is a pattern the car of the pattern must be an atom.

The system does not perform truth maintenance; i.e., axioms entered into the data base due to a forward-chaining axiom are not removed when the axiom is removed.

9.2 Examining Forward Production Axioms

(printf 'form) and (printfq form)

These functions pretty print all axioms whose triggers are specified by the form argument, which can be either a predicate name or a pattern. If it is a predicate name, all forward-chaining axioms with the given trigger name will be printed. Otherwise all forward-chaining axioms whose trigger matches with the given pattern will be pretty printed. Note that if the form is a pattern the car of the pattern must be an atom.

(printc 'form) and (printcq form)

These functions pretty print all axioms whose conclusions are specified in the form argument. The form argument can be either a predicate name or a pattern. If it is a predicate name then all forward-chaining axioms that have as a member of thier conlusion list an atomic formula with the given predicate name will be pretty printed. Otherwise all forward-chaining axioms who have a member of their conclusion list that unifies with the given pattern will be pretty printed.

(triggers)

Returns a list of all the predicate names which are trigger names for forward-chaining axioms.

9.3 Tracing Forward Chaining

Because the forward-chaining mechanism is defined in HORNE, the standard tracing functions (e.g., htraceall) are useable for debugging forward-chaining axioms. In addition, the following trace facilities are provided.

(trace-assertions)

This causes the system to print out all axioms that are asserted by the forward chaining system. The system default is that this tracing is on.

(untrace-assertions)

Stops the tracing of assertions made by the forward chaining system.

(trace-forward)

Causes the system to print out the trigger and rule of any forward-chaining axiom that has been triggered.

(untrace-forward)

Undoes the effects of "trace-forward".

9.4 I/O

I/O for forward production rules are handled by the I/O functions documented in Section 6 (Saving and Restoring Programs). An exception is the function "save-indices" which cannot be used to save forward chaining rules.

9.5 Editing Forward Chaining Axioms

(edit-forward 'predname)

The above call will get you into an interactive editor for forward-chaining axioms. The options are listed as below.

a "assert"

Prompts for a position of the new axiom. If correctly specified it prompts for axiom. Enter axiom as a list of its components.

r "retract"

Prompts the user for the number of an axiom to remove. Will return error message if number is not properly specified.

p "print"

Pretty prints out the current version of the axiom list.

u "undo"

Undoes any and all changes made in the current edit session.

q "quit"

Leaves the editor and makes all changes.

9.6 Examples

The first example shows the use of forward chaining for a simple equality system. The rules capture the transitivity and symmetric properties of equality. The rules are:

```
(addf 'all '((MYEQ ?y ?z)) 'p '((MYEQ ?y ?x) (MYEQ ?x ?z)))
(addf '(MYEQ ?s ?d) '((MYEQ ?d ?s)) 'p '())
```

If we now add

```
(addaq ((MYEQ w e) k))
```

the following axioms are also asserted by the system:

```
((MYEQ e w) p)
((MYEQ w w) p)
((MYEQ e e) p)
```

If we now add:

```
((addaq ((MYEQ r e) k))
```

then the following are also asserted:

```
((MYEQ e r) p)
((MYEQ r w) p)
((MYEQ w r) p)
```

The second example involves forward chaining rules that are used to maintain consistency in a data base for a simple blocks world. Here the chaining rules call LISP functions to delete axioms.

```
(addf '(pickup ?d) '((holding ?d)
                     (RETRACT (ontable ?d))
                     (RETRACT (clear ?d))
                     (RETRACT (handempty)))

                   'index

                   '((ontable ?d)
                     (clear ?d)
                     (handempty)))

(addaq ((ontable block1) k)
       ((clear block1) k)
       ((handempty) k))
```

If we now add

```
(addaq ((pickup block1) k))
```

then the axiom ((holding block1) index) becomes true and the predicates (ontable block1) (clear block1) and (handempty) are deleted from the data base.

10. BUILT-IN PREDICATES

This section documents the built-in predicates that are already defined in HORNE.

(ASSERT <axiom>)

Adds the specified axiom to the data base at the end of the axiom list for the specified predicate. Thus, this performs a similar function to addz but is callable from HORNE and returns t. All logic variables in the new axiom that are bound in the current environment will be replaced by their values before the new axiom is added.

(ATOM <term>)

Succeeds if *<term>* is an atom.

(BOUND ?x)

Succeeds only if ?x is not a variable. It succeeds on any other non-grounded term. For example, *(bound (f ?x))* succeeds. Equivalent to but faster than *(UNLESS (VAR ?x))*.

(DISTINCT <term$_1$> <term$_2$>)

Succeeds if both terms are fully grounded, but to different atoms. If a term is not fully grounded, this posts a constraint on the variable(s) and succeeds.

(EQ <term$_1$> <term$_2$>)

Succeeds if <term$_1$> equals <term$_2$> (i.e., they unify) (see Section 8.1).

(FAIL)

This predicate is always false.

(GENVALUE <variable> <LISP expression>)

Sets the HORNE variable <variable> to first value in list returned by evaluating the <LISP expression>. Other values are used for backtracking (see Section 5.1).

(GROUND <term$_1$>)

Succeeds if term$_1$ is a fully grounded term, i.e., it contains no variables.

(IDENTICAL <term$_1$> <term$_2$>)

Succeeds if <term$_1$> and <term$_2$> are structurally identical; i.e., if they unify without the equality mechanism.

(MEMBER <term$_1$> <list>)

Succeeds if <term$_1$> is equal (i.e., HORNE equality) to a term in the list.

(NEQ <term$_1$> <term$_2$>)

Succeeds if both <term$_1$> and <term$_2$> are fully grounded, but to different values. Otherwise it fails.

(RETRACT <term$_1$>)

Retracts all axioms whose head unifies with <term$_1$>

(RPRINT <term$_1$> ... <term$_n$>)

The values of <term$_1$> through <term$_n$> are printed on successive lines.

(RTERPRI)

Prints a line feed.

(SETVALUE ⟨variable⟩ ⟨LISP expression⟩)

Sets the HORNE variable *⟨variable⟩* to the value of the LISP expression *⟨LISP expression⟩*. Any logic variables in *⟨LISP expression⟩* are replaced by their logic bindings before LISP evaluation (see Section 5.1).

(UNLESS ⟨atomic formula⟩)

Succeeds only if the call *(proveq ⟨atomic formula⟩)* fails. This gives us proof by failure. Note that variables change in interpretation in the UNLESS function; e.g., if we are given the fact that (P A) is true, then

(UNLESS (P B)) will succeed.

(UNLESS (P A)) will fail as expected.

But (UNLESS (P ?x)) also fails, since (P ?x) can be proven.

(VAR ⟨variable⟩)

Succeeds only if *⟨variable⟩* is an unbound variable.

/ The cut symbol. It has no effect until HORNE tries to backtrack past it, and then the prover immediately fails on the subproblem it was working on. An alternate definition: cut always succeeds, and when executed, removes all choice points in the proof from the point when the axiom containing the cut was selected to the current point of the proof.

11. HASHING

A hashtable can be declared for a predicate name whether it currently has axioms asserted for it, or will have axioms asserted later. It can also be used to redefine an already existing hashtable for the predicate. The hashtable allows the axioms for a predicate to be stored according to the values of the arguments to the predicate. They can currently only be used on argument positions that do not allow equality reasoning. For example, consider a one-place predicate P with hashing on its argument into three buckets. If we have asserted the facts (P A), (P B), (P C), (P D), (P (f A)) and (P (g ?x)), the hashed structure might look like the following (ignoring efficiency encodings):

P
- bucket 1 → (P A)
- bucket 2 →(P B), (P D)
- bucket 3 → (P C)
- function bucket → (P (f A)), (P (g ?x))
- variable bucket → (P A), (P B), (P C), (P D), (P (f A)), (P (g ?x))

Now if we query (P A), we would hash on A to bucket 1 and just unify (P A) with those axioms there, i.e., only (P A). Similarly, for (P E), if hashing on E gives bucket 3, then (P E) would be unified only with (P C). Any complex argument, such as (P (g B)), will be checked against the special *function bucket*, i.e., (P (f A)) and (P (g ?x)). Finally, any query with a variable, e.g., (P ?y), will be matched against the *variable bucket* which contains the complete axiom list.

As one can see, if equalities were allowed on terms in the argument position, this structure might fail. For example, given B = F, if we query (P F), and hashing on F gives bucket 1, then (P F) will be checked only against (P A) and would fail. Hashing with equality is being considered for the next version.

Hash tables are defined as follows:

(define-hashtable <predicate name>)

For forward chaining axioms, the trigger can be hashed using the function

(define-hashed-trigger <predicate name>).

For both of these uses, the system then prompts for paths through a formula to where the hashing should take place, and for the size of the buckets for each hash. The simple options for paths are as follows:

<number>

Hash on n^{th} argument to predicate.

(i <number>)

Hash on first atom found by successively taking CARs on the n^{th} argument to predicate.

Arbitrary paths may be built by specifying a sequence of CARs and CDRs starting from the predicate name. Thus the path *(CAR CDR)* is equivalent to the first argument. The path *(CAR)* would give the predicate name. The only other possibility in a path is to specify an arbitrary number of CARs, specified as CAR* in the path. Thus entering *(CAR* CDR CDR)* is equivalent to *(i 2)*.

The minimum number of buckets in a hashtable is 3; one for variables, one for lists (i.e., functions), and one for atoms. The number of buckets for atoms is the only size under programmer control. Thus, entering a 5 when prompted will produce 5 buckets for atoms.

A sample session that hashes a predicate **MYPRED** on the form of its second argument (into 10 buckets), and on some other arbitrary position in the third argument (into five buckets) follows:

→*(define-hashtable MYPRED)*
Enter path spec: *2*
Hashtable size? ("q" to respecify path) *10*
Enter path spec: *(CAR* CAR CDR CAR CDR CDR)*
Hashtable size? ("q" to respecify path) *5*
Enter path spec: *q*
Hashtable defined.

12. CONTROLS ON HORNE

The following global variables affect the behavior of HORNE:

H$$LIMIT

The number of steps HORNE can take before asking the user whether it should continue. Default value is 500. To continue, simply enter *y*, to terminate enter *n*. You can enter debug mode by entering *d*, after which typing *go* gets you back to the question whether to continue.

H$$PARTITION$CHK

The mechanism that adds information to the TYPE matrix does extensive consistency checking involving XSUBTYPEs. If no XSUBTYPE axioms are present the consistency testing is wasted. If this flag is set to nil then the testing is turned off. Default value is "t".

TESTFLAG

If this flag is set to nil then consistency will not be checked when TYPE axioms are added. Default is non nil.

The following functions also control the behavior of HORNE:

(warnings)

Enables the printing of warning messages at the user's terminal. By default, warning messages are printed.

(nowarnings)

Disables the printing of warning messages. By default, warning messages are printed.

13. EXAMPLES

13.1 A Simple Example

The following is a simple session with HORNE:

```
*(addzq ((HAPPY ?person ?item) <
            (DESIRABLE ?item)
            (CAN-AFFORD ?person ?item))
        ; you can afford items if you have money
        ((CAN-AFFORD ?person ?item) <
            (HAS-MONEY ?person))
        ; but love is for free
        ((CAN-AFFORD ?person Sweetheart) <)
        ((DESIRABLE Newsuit) <)
        ((DESIRABLE Caviar) <)
        ((DESIRABLE Sweetheart) <)
        ((HAS-MONEY Sam))    )

*(htraceall)
; prove JOHN can be happy even if he has no money
*(proveq (HAPPY JOHN ?why))

(q-1) (HAPPY JOHN ?why)
      (q-2) (DESIRABLE ?why)
      (r-2) (DESIRABLE Newsuit)
      (q-2) (CAN-AFFORD JOHN Newsuit)
            (q-3) (HAS-MONEY JOHN)
            ; note, backtracking to (q-2) (DESIRABLE ?why)
      (r-2) (DESIRABLE Caviar)
      (q-2) (CAN-AFFORD JOHN Caviar)
            (q-3) (HAS-MONEY JOHN)
            ; backtracking again to (q-2) (DESIRABLE ?why)
      (r-2) (DESIRABLE Sweetheart)
      (q-2)(CAN-AFFORD JOHN Sweetheart)
      (r-2) (CAN-AFFORD JOHN Sweetheart)
(r-1) (HAPPY JOHN Sweetheart)
; end of trace, the value returned is:
((HAPPY JOHN Sweetheart))
```

13.2 The Same Example with Posting

```
*(addzq ((HAPPY ?person ?item) <
              (POST (DESIRABLE ?item))
              (CAN-AFFORD ?person ?item))
         ((CAN-AFFORD ?person ?item) <
              (HAS-MONEY ?person))
         ((CAN-AFFORD ?person Sweetheart))
         ((DESIRABLE Newsuit) <)
         ((DESIRABLE Caviar) <)
         ((DESIRABLE Sweetheart) <)
         ((HAS-MONEY Sam))    )

*(htraceall)

*(proveq (HAPPY JOHN ?why))

(q-1) (HAPPY JOHN ?why)
      (q-2) (POST (DESIRABLE ?why))
      (r-2) (POST (DESIRABLE (any ?0006 ((DESIRABLE ?0006)))))
      (q-2) (CAN-AFFORD JOHN (any ?0006 ((DESIRABLE ?0006)))))
            (q-3) (HAS-MONEY JOHN)
            ; in trying the second axiom for CAN-AFFORD, we must
            prove (DESIRABLE Sweetheart) to unify Sweetheart
            with (any ?0006 ...)
      (r-2) (CAN-AFFORD JOHN Sweetheart)
(r-1) (HAPPY JOHN Sweetheart)
((HAPPY JOHN Sweetheart))
```

The only difference between this proof and the proof in 13.1 is when the predicate DESIRABLE is proved. In the first, we would backtrack through all values until one was found that succeeded. In the second, the rest of the proof is done first, and then when a value for ?why is found, it is checked to see if we can prove it is DESIRABLE.

13.3 An Example Using Types

This example uses a type hierarchy with two types, PROFESSOR and MUSICIAN, that intersect with the subtype MUSICAL-PROFESSOR.

```
; The type hierarchy
```

```
(addzq ((ISUBTYPE PROFESSOR PEOPLE))
       ((ISUBTYPE MUSICIAN PEOPLE))
       ((INTERSECTION MUSICAL-PROFESSOR
                      PROFESSOR
                      MUSICIAN)))
```

; The axioms:

all professors teach, and all musicians sing
someone is happy if they teach and sing

```
(addzq ((TEACH ?p:PROFESSOR))
       ((SING ?m:MUSICIAN))
       ((HAPPY ?p) < (TEACH ?p) (SING ?p)))
```

; Here we could add hundreds of professors and musicians, and a few musical-professors.

```
(addzq ((ITYPE JACK MUSICAL-PROFESOSR)))
```

Now we can prove the following:

Is Jack Happy? yes.

```
(proveq (HAPPY JACK))

(q-1) (HAPPY JACK)
      (q-2) (TEACH JACK)
      (r-2) (TEACH JACK)
      (q-2) (SING JACK)
      (r-2) (SING JACK)
(r-1) (HAPPY JACK)
```

Who is happy? All musical professors.

```
(q-1) (HAPPY ?x)
      (q-2) (TEACH ?x)
      (r-2) (TEACH ?y:PROFESSOR)
      (q-2) (SING ?y:PROFESSOR)
      (r-2) (SING ?z:MUSICAL-PROFESSOR)
(r-1) (HAPPY ?z:MUSICAL-PROFESSOR)

((HAPPY ?z:MUSICAL-PROFESSOR))
```

INDEX OF FUNCTIONS

REFERENCES

Bowen, K. A. PROLOG. In *Proceedings of the ACM Annual Conference.* October, 1979.

Foderaro, J. K. *The FRANZ LISP Manual.* 1980.

Kornfeld, Wm. A. Equality for Prolog. In *Proceedings of 8th IJCAI.* Karlsruhe, W. Germany, August 1983.

Kowalski, R. A. Predicate logic as programming language. In *Proceedings of IFIP 74.* Amsterdam: North Holland. 1974.

Kowalski, R. A. *Logic for Problem Solving.* New York: Elsevier-North Holland, 1979.

Winston, P. H. and Horn, B. K. H. *LISP.* Reading, MA: Addison-Wesley, 1981.